Exact String Matching: KMP & Boyer-Moore Algorithms

Chin Lung Lu

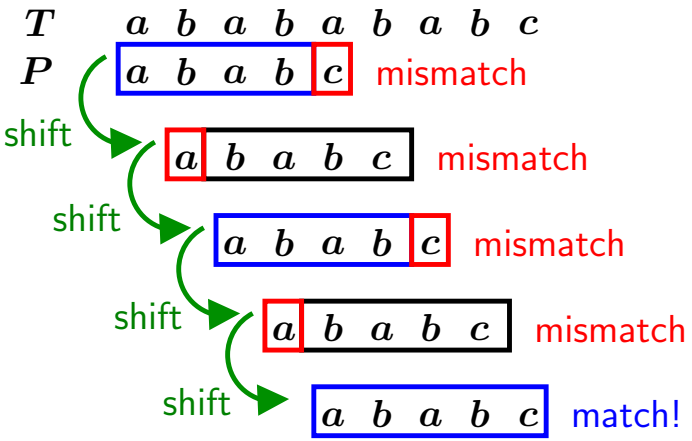
Computational Biology

Analyses and Applications of Sequences

Exact String Matching

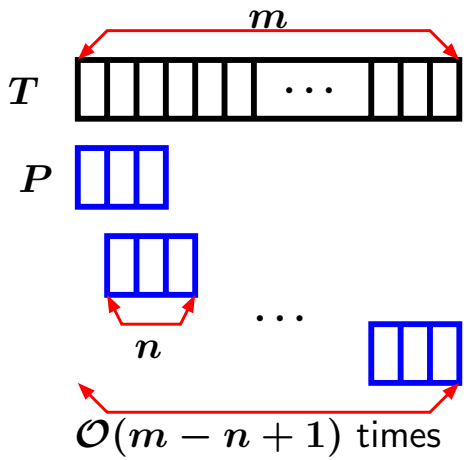
- Given two strings T and P, where $|T| = m$ and $|P| = n$, find all the occurrences of P in T?
- (**Simplified version**) Given two strings T and P, if P occurs in T?
- Naive method: $\mathcal{O}(mn)$
- KMP algorithm (Knuth, Morris and Pratt, 1977): $\mathcal{O}(m+n)$
- Boyer-Moore algorithm (Boyer and Moore, 1977): $\mathcal{O}(m+n)$

Naive Method



- Simple, but not efficient because it cannot avoid rescanning T

Naive Method



- Time complexity of naive method: $\mathcal{O}(mn)$

How to Speedup Brute-Force?

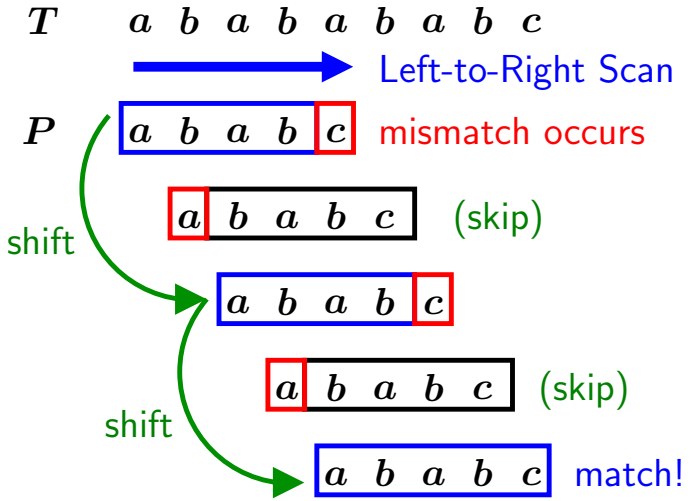
- Shift P by more than one place when a mismatch occurs without missing any occurrence of P in T

1. KMP algorithm
   - Proposed by Knuth, Morris and Pratt at 1977
   - Time complexity: $\mathcal{O}(m+n)$
2. Boyer-Moore algorithm
   - Proposed by Boyer and Moore at 1977
   - Time complexity: $\mathcal{O}(m+n)$

KMP Algorithm

- Left-to-right scan
- Failure function shift rule

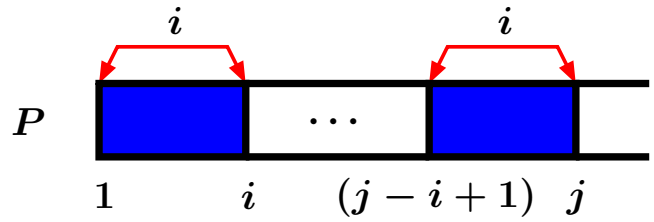
KMP Algorithm: A Scenario



Failure Function

- If $P = p_1 p_2 \cdots p_n$, then its **failure function** f is defined as follows, where $1 \le j \le n$.

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } & \text{if such an} \\ p_1 \cdots p_i = p_{j-i+1} \cdots p_j, & i \ge 1 \text{ exists,} \\ 0, & \text{otherwise.} \end{cases}$$



Examples of Failure Function

- **Example 1**:

P	a	b	a	b	c
$f(j)$	0	0	1	2	0

- **Example 2**:

P	a	b	a	c	a	b	a	b
$f(j)$	0	0	1	0	1	2	3	2

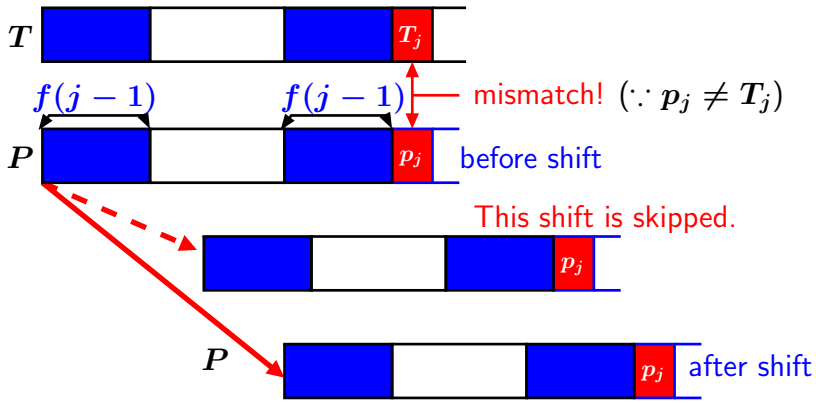
KMP Algorithm

/* $T = T_1T_2 \cdots T_m$ and $P = p_1p_2 \cdots p_n$ */

1. $i = 1$; $j = 1$;
2. **while** $i \leq m$ and $j \leq n$ **do**
3. **if** $T_i = p_j$ **then**
4. $i = i + 1$; $j = j + 1$;
5. **else if** $j = 1$ **then** $i = i + 1$;
6. **else** $j = f(j - 1) + 1$;
7. **end if**
8. **end while**
9. **if** $j = n + 1$ **then** "a match" **else** "no match".

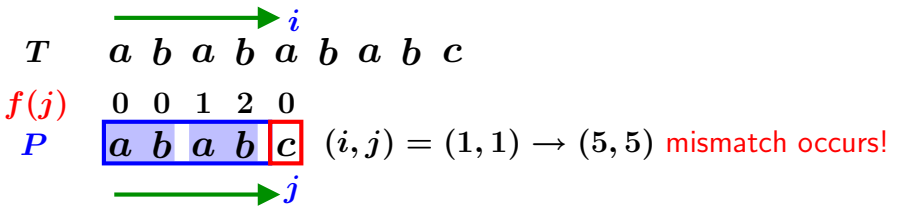
- KMP algorithm costs $\mathcal{O}(m)$ time, if $f(j)$ for all $1 \leq j \leq n$ is known in advance. (why?)

KMP Algorithm: Basic Idea

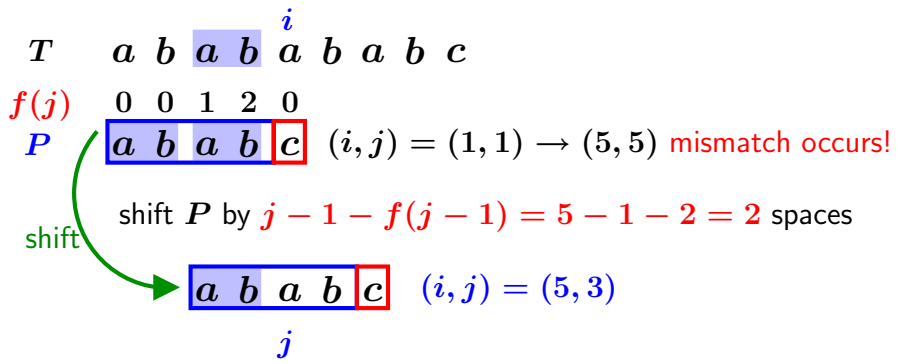


- P is shifted by $j - 1 - f(j - 1)$ places.

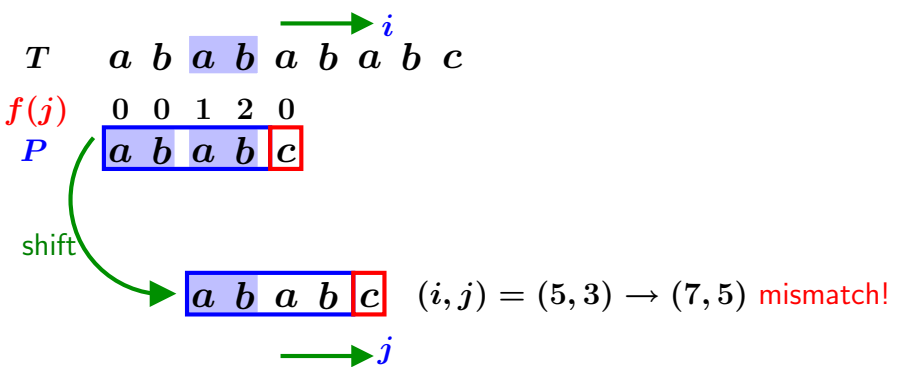
KMP Algorithm: An Example ❶



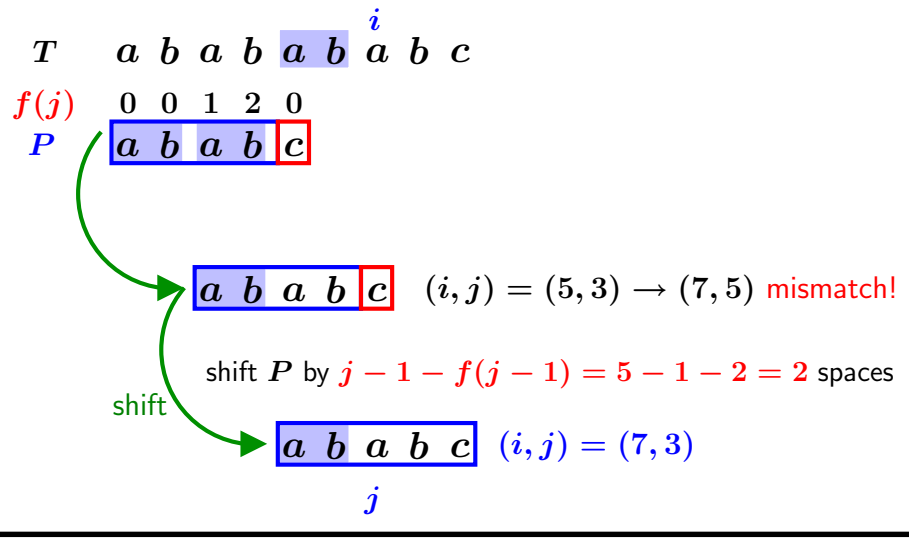
KMP Algorithm: An Example ❷



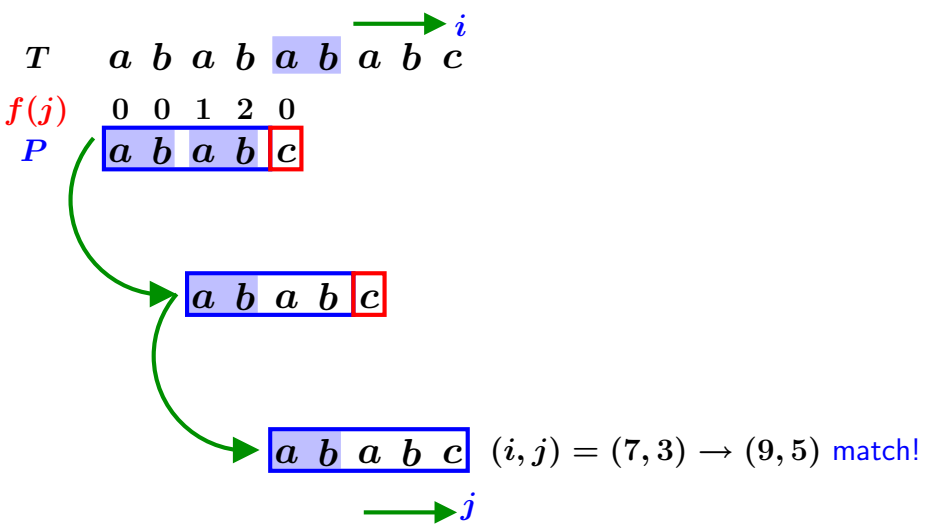
KMP Algorithm: An Example ❸



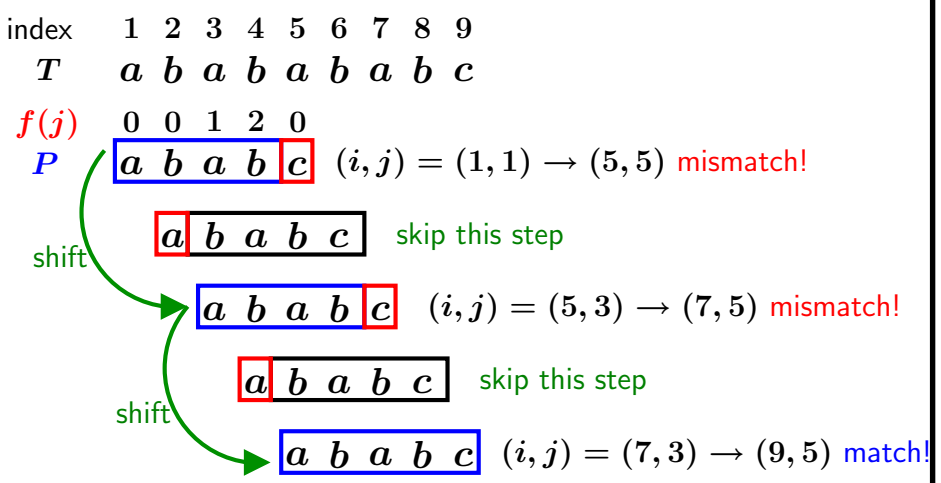
KMP Algorithm: An Example ❹



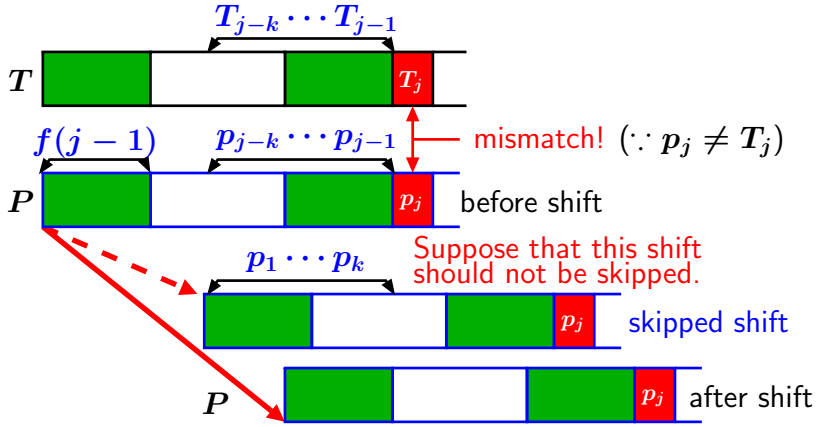
KMP Algorithm: An Example ❺



KMP Algorithm: An Example



Correctness of KMP Algorithm

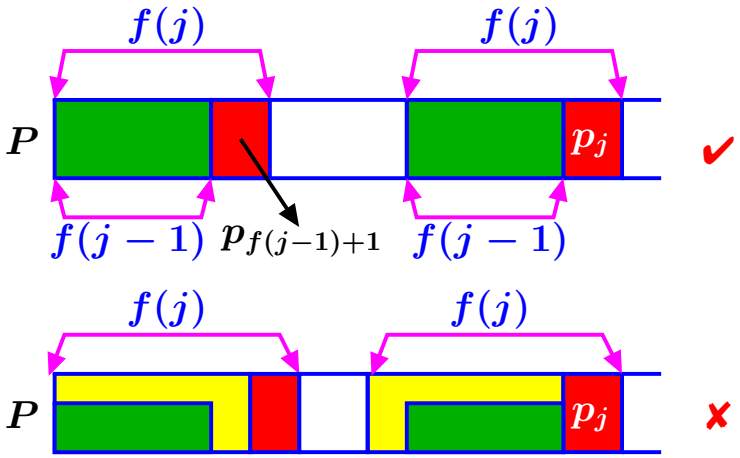


Then $p_1 \cdots p_k = T_{j-k} \cdots T_{j-1} = p_{j-k} \cdots p_{j-1}$

Hence $f(j-1) = k > f(j-1)$, a contradiction.

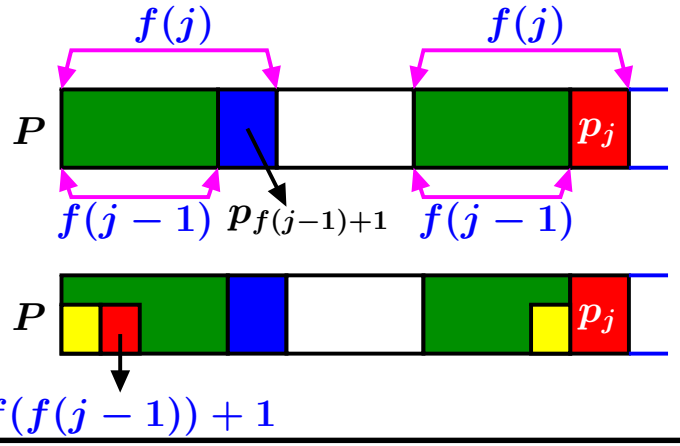
How to Compute Failure Function?

- If $p_{f(j-1)+1} = p_j$, then $f(j) = f(j-1) + 1$.

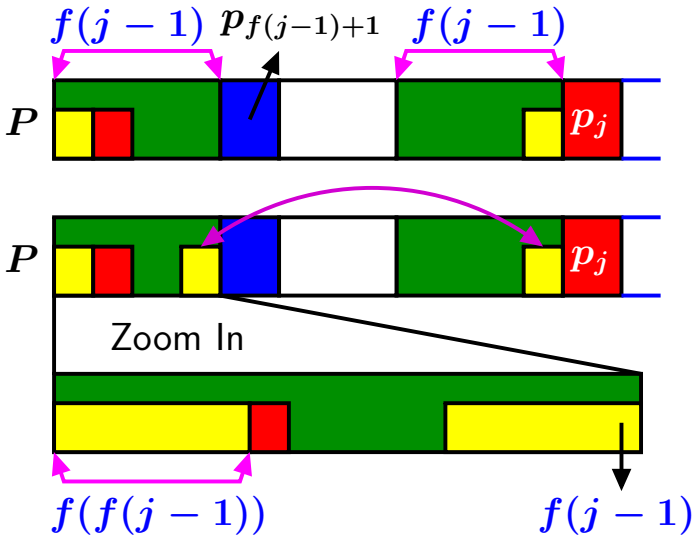


How to Compute Failure Function?

- If $p_{f(j-1)+1} \neq p_j$ and $p_{f(f(j-1))+1} = p_j$, then $f(j) = f(f(j-1)) + 1$.



How to Compute Failure Function?



Failure Function Algorithm

/* Compute the failure function for $P = p_1 p_2 \cdots p_n$ */

```
1.   f(1) = 0;
2.   for j = 2 to n do
3.       i = f(j − 1);
4.       while p_j ≠ p_{i+1} and i > 0 do
5.           i = f(i);
6.       if p_j = p_{i+1} then
7.           f(j) = i + 1;
8.       else
9.           f(j) = 0;
10.  end for
```

- Total cost is $\mathcal{O}(n)$ time by amortized analysis. (?)

Amortized Analysis of Algorithm

Given a sequence S of operations $op_1, op_2, \ldots, op_n$, let $T(op_i)$ be the worst case time-complexity of each op_i, where $1 \leq i \leq n$.

- Non-amortized analysis: the time-complexity of S is $T(S) = \sum_{i=1}^{n} T(op_i)$.
  - The opreations are independent of one another.
- Amortized analysis: $T(S) << \sum_{i=1}^{n} T(op_i)$
  - You cannot spend money more than that saved in your account of the bank.
  - We work hard when we are young, we reap when we are old.

Complexity of KMP Algorithm

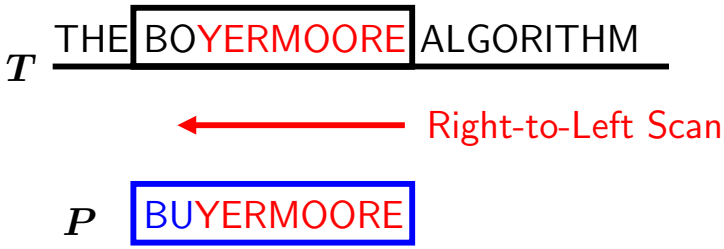
- KMP algorithm costs $\mathcal{O}(m)$ time, if $f(j)$ for all $1 \leq j \leq n$ is known in advance. (why?)
  - "$i = i + 1$" (i.e., i moves right on T) is executed for a total of at most m times, since i never moves left and $|T| = m$.
  - Totally, "$j = j + 1$" (i.e., j moves right on P) is exccuted at most m times
  - As a result, "$j = f(j-1) + 1$" (i.e., j moves left on P) is totally executed at most m times, as otherwise $j < 0$ (i.e., j must fall off the left end of P).

Boyer-Moore Algorithm

- Right-to-left scan
- Bad character shift rule
- Good suffix shift rule

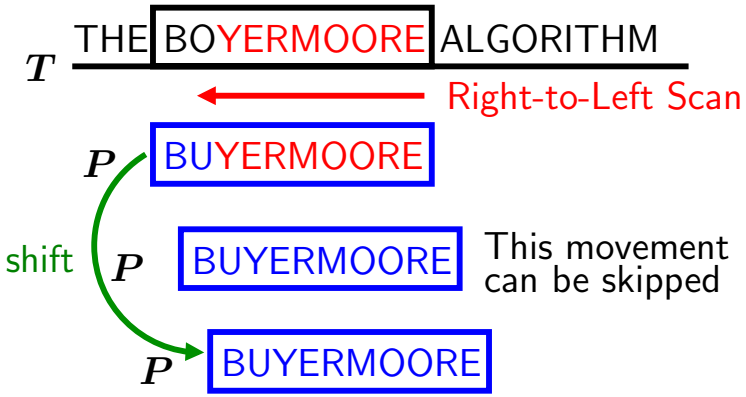
Right-to-Left Scan

- Check whether P occurs in T at some position in the right-to-left scaning manner



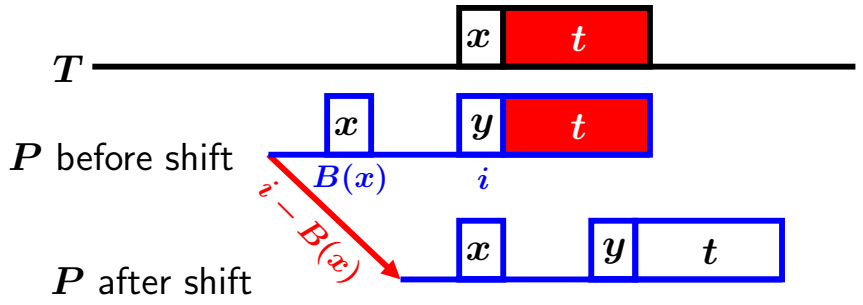
Bad Character Shift Rule

- What happened if the initial mismatch occurs?

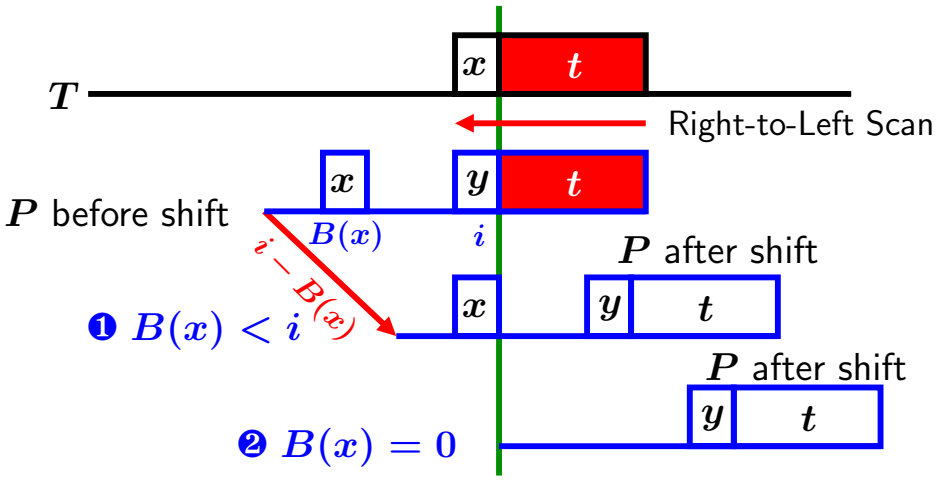


Bad Character Shift Rule

- Bad character rule: If $x \neq y$, then P is shifted right by $\max\{1, i - B(x)\}$ places.
  - $B(x)$: the position of right-most occurrence of x in P ($B(x) = 0$ if x does not occur in P)

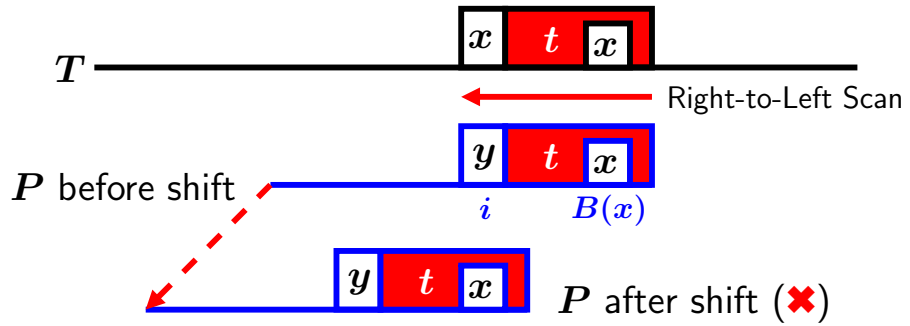


Bad Character Shift Rule



Bad Character Shift Rule

- If $B(x) > i$, then bad character rule has no effect.



Bad Character Shift Rule

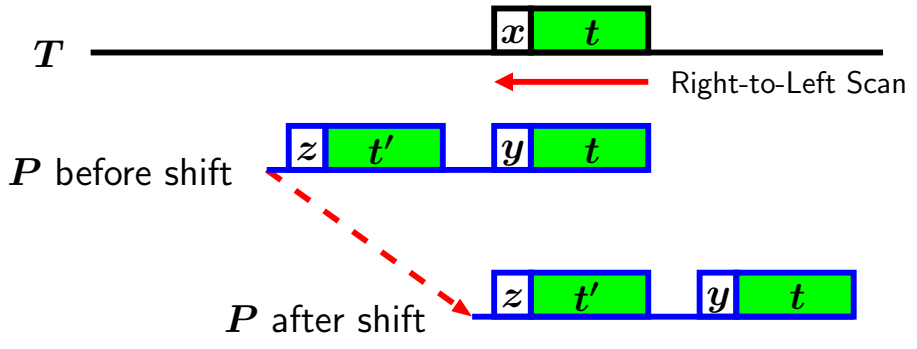
- How to compute $B(x)$ for all x in P? ($n = |P|$)

1. **for** $i = 1$ to n **do**
   $B(P[i]) = 0$;
   end for
2. **for** $i = n$ to 1 **do**
   if $B(P[i]) = 0$ **then** $B(P[i]) = i$;
   end for

	i	1	2	3	4	5	6	7	8	9	10
	$P[i]$	B	O	Y	E	R	M	O	O	R	E
1.	$B(P[i])$	0	0	0	0	0	0	0	0	0	0
2.	$B(P[i])$	1	8	3	10	9	6	8	8	9	10

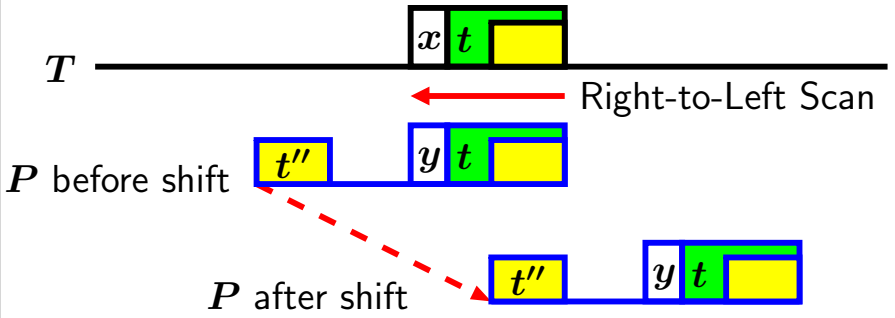
Good Suffix Rule: Case ❶

- If $x \neq y$, then find the right-most copy t' of t in P such that t' is not a suffix of P and $z \neq y$



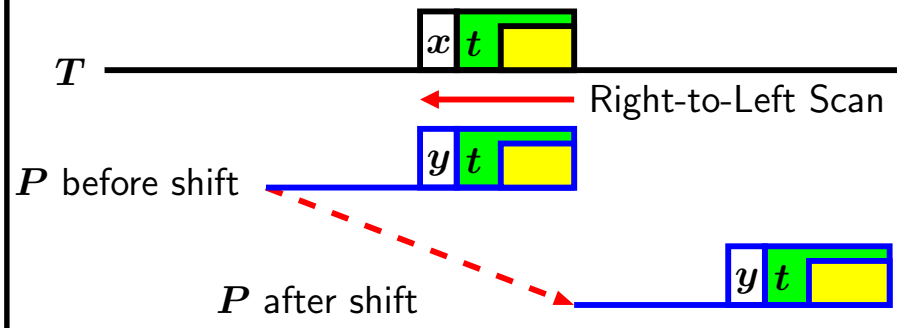
Good Suffix Rule: Case ❷

- If t' does not exist, then find the largest prefix t'' of P such that it is equal to a suffix of t



Good Suffix Rule: Case ❸

- If t' and t'' do not exist, then



Good Suffix Rule: Summary